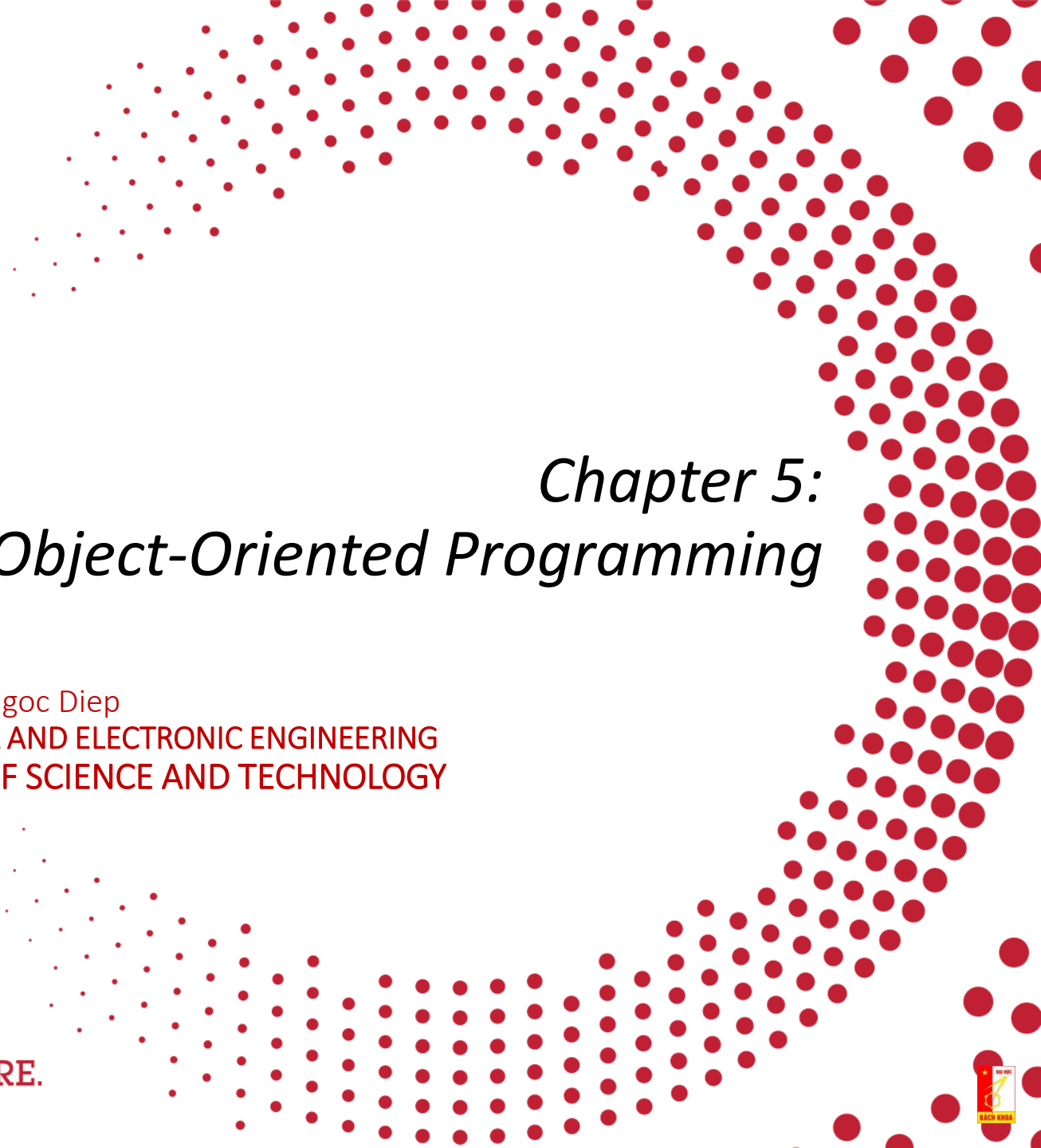


HUST

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

C/C++ Programming Techniques
ET2031/ ET2031E

ONE LOVE. ONE FUTURE.



Chapter 5: Object-Oriented Programming

Lecturer: PhD. DO Thi Ngoc Diep
SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

- 5.1. Introduction to C++
- 5.2. Overview of OOP
- 5.3. Class and Object in C++
 - Constructors and Destructor
 - Method and Property Members of Object
 - Operator overloading

Method and Property Members

Function vs. Method

- In structural programming: data passed to the function as an argument
- OOP: **Object** becomes the subject of the program

```
struct Student {  
    char name[20];  
    int major;  
};  
void assign (Student &sv, int  
room)    { ... }  
void check (Student &sv)  
    { ... }
```

```
Student sv = {"Nguyen A", 62 };  
assign(sv, 103);  
check(sv);
```

```
class Student {  
    char name[20];  
    int major;  
    void assign(int room)  
        { ... }  
    void check()  
        { ... }  
    ...  
};  
void main() {  
    Student sv;  
    sv.setName("Nguyen A");  
    sv.setMajor(62);  
    sv.assign(103);  
    sv.check();  
}
```

- Distinguish between a member function and a free/normal function
 - Member function (method): belongs to a class, and will also belongs to objects of that class
 - Free/normal function: functions defined outside of classes, which are sub functions in C
 - In C++, member function definitions can be placed inside or outside the class

Member functions/ Methods

```
class Point {
    float _x, _y;
public:
    void setXY(float x, float y);           //declaration
    float getX(){ return _x; }             //definition
    float getY(){ return _y; }             //definition
    float distanceTo(Point p);              //declaration
};

void Point::setXY(float x, float y){
    _x = x;
    _y = y;
}

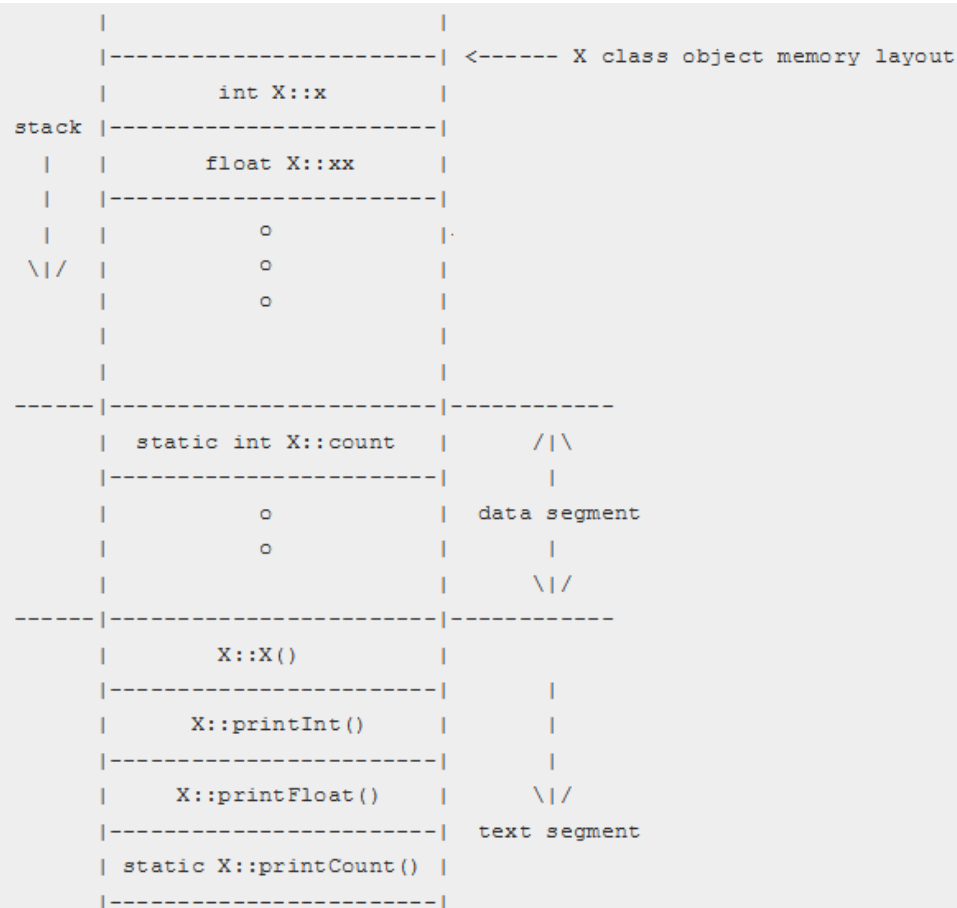
float Point::distanceTo(Point p){
    float d = (p._x-_x)*(p._x-_x) + (p._y-_y)*(p._y-_y);
    return sqrt(d);
}

int main()
{
    Point p1, p2;
    p1.setXY(10,10);
    p2.setXY(20,20);
    cout<<"D="<<p1.distanceTo(p2)<<endl;
    return EXIT_SUCCESS;
}
```

Static members

- A static property: exists only once for all objects of a class (even if no objects have been created yet)
 - Must initialize out of the class
- Static method: used to handle static properties of the class

```
class X {  
    int      x;  
    float    xx;  
    static int count;  
  
public:  
    X() {...}  
    void printInt() {...}  
    void printFloat() {...}  
    static void printCount() {...}  
};  
  
int X::count = 0;
```



- A static property: exists only once for all objects of a class (even if no objects have been created yet)
 - Must initialize out of the class
- Static method: used to handle static properties of the class

```
class C {  
public:  
    static int count;  
    C()      { count++; }  
    ~C()     { count--; }  
    static int getCount()  
        { return count; }  
    ...  
};  
int C::count = 0;
```

```
C c1, c2, c3[10],  
  *c4 = new C(),  
  *c5 = new C[20];  
delete c4;  
  
cout << "Number of current C objects: "  
      << C::getCount() << endl;  
// cout << C::count << endl;  
// cout << c1.count << endl;  
// cout << c2.getCount() << endl;
```

- Can be accessed through the class name and "::" operator, or as members of objects as usual
- There is no 'this' pointer in static methods

- Constant method: in that method the properties of the class will be constant/can not be changed
 - Member properties cannot be assigned or changed in a constant method
 - Cannot call non-constant methods inside a constant method
 - Should declare all methods that do not change member properties as constants
 - Constant object: only its constant methods can be used

```
class Circle {  
    ...  
    void setR(double r) { this->r = r; }  
    double getR() const { return r; }  
    double area() const { return PI*r*r; }  
    void cf(double r) const {  
        area();          // OK  
        setR(r);         // error  
        this->r = r;      // error  
    }  
};
```

```
Circle c1;  
const Circle c2(2.333);  
  
c1.setR(1.22);      // OK  
c2.setR(1.22);      // error  
  
c1.area();           // OK  
c2.area();           // OK  
  
cout << c1.getR()    // OK  
     << c2.getR();   // OK
```

- *friends of a class* can access the private members of that class

```
class Circle {  
private:  
    double r;  
public:  
    friend void printCircle(Circle c);  
    friend class Ellipse;  
};  
  
void printCircle(Circle c)  
    { cout << "Radius: " << c.r; }  
  
class Ellipse {  
private:  
    double rx, ry;  
public:  
    void convert(Circle c) {  
        rx = ry = c.r;  
    }  
};
```

- Friend function is not a method of class
- Friend function call is the same as a regular function call
- If class A is declared as a friend of class B, then all methods of class A can access private members of class B

- Multiple functions in the same scope can have the *same name*
 - but must be *different in input parameters* (number of parameters, type of each parameter)

```
1. int  compare(int n1, int n2);
2. int  compare(float x1, float x2);
3. bool compare(float x1, float x2); // error
4. int  compare(string& s1, string& s2);
5. int  compare(const string& s1, const string& s2);
```

- To determine the correct function call:
 - find the function that has the exactly same input parameters,
 - if not, check the conversion for type.

```
6. string ss1("xyz"), ss2("mpnq");
7. const string cs("aaa");
8. compare(1.3, 2.5); // error
9. compare("abcd", "12345"); // function 5 is called
10. compare(ss1, ss2); // function 4 is called
11. compare(ss1, cs); // function 5 is called
```

- The same

- ```
class C {
 public:
 int compare(int x, int y);
 int compare(float x, float y);
};
```

- [Note] Overloading in the subclass will hide the methods of the parent class with the same name

- ```
class D : public C {  
    public:  
        int compare(string s1, string s2);  
};
```

```
D d;  
d.compare("1234", "abcd");    // OK  
d.compare(10, 20);            // error  
d.C::compare(10, 20);         // OK
```

- Function parameters can have default values (used if omitted in the function call)
- Default parameters must be the last parameters of the function
 - `void out(double x, int width = 7, int prec = 3) {...}`
`out(1.2345, 10, 5);`
`out(1.2345, 10);` `// → out(1.2345, 10, 3);`
`out(1.2345);` `// → out(1.2345, 7, 3);`
 - `void f(char c = 'y', int n, bool b = true) {...}` `// error`
 - `void temp(int i=10, float j=8.8);`
`temp(3.4);` `// → temp(3, 8.8);`

- can be declared in prototype and omitted in the definition
 - ```
double df(double x, int order = 1);
// ...
double df(double x, int order) {...}
```
- Expressions can be used as default values, but cannot contain other parameters of the function
  - ```
UserProfile usr;  
double out(double x, int prec = getPrecOption(usr));  
double next(double x, double dx = diff(x)); // error
```

- Avoid confusion with overlapping functions
 - `void input(double& x);`
`void input(double& x, const char* prompt = "Nhap so: ");`
`double y = 0.5; input(y);` // error - ambiguous
- Method with default parameter: the same
 - `class Vehicle {`
`void out(int prec = 3);`
`};`

- Default parameter for constructor

```
• enum class Color {  
    black, red, white  
};  
class Vehicle {  
    public:  
        Vehicle() { ... }  
        Vehicle(Color c = Color::black, int wheels = 4) { ... }  
};  
Vehicle v1(Color::red);  
Vehicle v2(Color::white, 8);  
Vehicle v3; // error
```

Operator overloading

- Operators in C++ can be re-defined for user-defined types

- For example: Define +, -, * operators for a Vector class:

```
Vector v1, v2, v3;  
v3 = -v1 + v2*2; // using 4 operators
```

- Operators for basic types is built-in and cannot be re-defined:

```
int x = 3 + 2*5;  
double y = 2.54/1.23 + 3.11;
```

- A few operators cannot be redefined

```
. .* :: ?: sizeof
```

- All operators defined in a class will be inherited, except for the assignment operator '='
- The order/precedence of the operators in the expression cannot be changed

- Define an operator function
 - can be as a global function or a method of a class
 - Syntax:

```
<return_type> operator <operator_symbol>(<type> <parameter>) {  
    ... // function body  
}
```
- Can not define default parameters for operator functions
- If defined as a global function, it is usually declared as friend of the class
- If defined in the class, as default the first parameter is always the called object

- as a global function with one parameter

```
class Vector {
    private:
        double x, y, z;
    public:
        ...
        friend Vector operator -(const Vector& v); // to use the private parameters
};

Vector operator -(const Vector& v) {
    return Vector(-v.x, -v.y, -v.z); }
```

- or a method of the class with no parameters

```
class Vector {
    private:
        double x, y, z;
    public:
        ...
        Vector operator -() const
            // the first parameter is always *this
            { return Vector(-x, -y, -z); }
};
```

Overloading for an unary operator

- Usage:
 - `Vector v1(1.2, 2.3, 4.5), v2;`
`v2 = -v1; // Can be used with both definitions above`
 - Or explicitly call the operator function :
 - `v2 = operator -(v1); // call global function`
- or:
- `v2 = v1.operator -(); // call method of the class`

Increment and Decrement operators (++ and --)

- Prefix and suffix operators
- The prefix operator: defined as usual
- The suffix operator: add a second parameter of type *int* (though not used).
- Example: definition in class (same for outside of the class)

```
class Check {  
    private:  
        int i;  
    public:  
        Check(): i(0) { }  
        Check operator ++ ()          Check operator ++ (int)  
        {                             {  
            Check temp;                Check temp;  
            temp.i = ++i;               temp.i = i++;  
            return temp;                return temp;  
        }                             }  
};
```

- Usage:

```
Check obj, obj1; obj1 = ++obj; obj1 = obj++;  
obj1 = obj.operator ++(); // call the prefix operator  
obj1 = obj.operator ++(0); // call the suffix operator
```

- Unary operator, no need to declare return type, defined in class only

```
class Fraction {  
private:  
    int a, b;  
public:  
    operator double() { return (double)a/(double)b; }  
    ...  
};
```

- Usage:

```
Fraction f(4, 5);  
double d = (double)f + 1.2;
```

- Note:

- Type conversion operator : conversion from class to another type
- Type conversion constructor : conversion from another type to class

- Use a global operator function with two parameters, or a method with one parameter in a class

- Example:

- ```
Vector operator -(const Vector& v1, const Vector& v2)
 { return Vector(v1.x-v2.x, v1.y-v2.y, v1.z-v2.z); }
```

or:

- ```
class Vector {
public:
    ...
    Vector operator -(const Vector& v) const
        // first parameter is *this
        { return Vector(x-v.x, y-v.y, z-v.z); }
};
```

- Usage:

- ```
v3 = v2-v1;
```

- Similar to unary operator:

- Often declared outside the class as friends to use hidden properties
  - Can explicitly call binary operator functions

- Example:

- ```
class Vector {  
    public:  
        bool operator ==(const Vector& v) const // in class  
        { return x == v.x && y == v.y; }  
        friend bool operator !=(const Vector&, const Vector&);  
};
```

- // out of class:

```
bool operator !=(const Vector& v1, const Vector& v2)  
    { return !(v1==v2); } // reuse == operator
```

- Other comparison operators can be similarly defined :

> < >= <=

- Assignment operator can only be defined in the class

```
class Marks{  
    private:  
        int m1; int m2;  
    public:  
        ...  
    void operator=(const Marks &M ) {  
        m1 = M.m1;  
        m2 = M.m2;  
    }  
}
```

- Other assignment operators can be similarly defined :

`+=    -=    *=    /=    ^=    &=    |=    <<=    >>=`

- Differ from other assignment operators:
 - Also known as the copy operator
 - If not declared, there is a default copy operator defined for the class with arguments of the same type to copy the member properties.
 - Not inherited by derived classes (hidden by default operator of subclasses)
- Distinguish from copy constructor
 - `Vector v2(v1), v3 = v2; // use copy constructor`
 - `v3 = v2; // assignment operator =`
- Distinguish from type conversion constructor
 - `string s1("12"), s2 = "ab"; // type conversion constructor`
 - `s2 = "xyz"; // assignment operator =`

- Used for dynamic memory allocation
- Attention: calling constructor/destructor is automatic, can't be interfered

```
class student{
    string name; int age;
public:
    student() { ... }
    student(string n, int a){
        cout << "Constructor 1 is called\n"; name = n; age = a;
    }
    void * operator new(size_t size){
        cout << "Overloading new operator is called\n";
        return malloc(size);
    }
    void operator delete(void * p){
        cout << "Overloading delete operator is called\n " << endl; free(p);
    }
};

int main(){
    student * p = new student("Yash", 24);
    delete p;
}
```

Overloading new operator is called
Constructor 1 is called
Overloading delete operator is called

Other special operators

- Function call operator: `p(x, y)`
- Index operator: `arr[i]`
- Comma operator: `a = (b=3, b+2);`
- Reference operator: `*ptr`
- Member access operator `.` or `->`
- Pointer member access operator `.*` or `->*`
- ...

- cout, cin are two objects of classes ostream and istream.
- The << and >> operators have been overloaded for I/O.
- Example: (★)
 - ostream& operator <<(int x) {...}
 - ostream& operator <<(float x) {...}
 - ostream& operator <<(double x) {...}
 - ostream& operator <<(char x) {...}
 - ostream& operator <<(const char* s) {...}
 - ...
 - istream& operator >>(int& x) {...}
 - istream& operator >>(float& x) {...}
 - istream& operator >>(double& x) {...}
 - istream& operator >>(char& x) {...}
 - istream& operator >>(char* s) {...}
 - ...

* The examples here are for illustrative purposes only. Actually the ostream and istream classes are not defined exactly like here. See more in the section about STL.

- The << and >> operators

```
class Vector {  
    // declare friend for operators ...  
};  
  
ostream& operator <<(ostream& s, const Vector& v) {  
    s << '(' << v.x << ", " << v.y << ", " << v.z << ')';  
    return s;  
}  
istream& operator >>(istream& s, Vector& v) {  
    s >> v.x >> v.y >> v.z;  
    return s;  
}
```

- Usage:

- Vector v1, v2;
 cout << "v1 = " << v1;
 cin >> v2;

1. Định nghĩa đầy đủ các toán tử cho lớp Vector 3 thành phần: cộng, trừ, nhân với số, tích vô hướng và có hướng
2. Định nghĩa các toán tử cho lớp String: + (cộng chuỗi hoặc ký tự), chuyển kiểu, xuất/nhập, [] (lấy phần tử)
3. Viết một lớp Array cho mảng động với các toán tử: += (thêm phần tử, nối hai mảng), [], chuyển kiểu

1. Define operators for 3-component Vector class: addition, subtraction, multiplication with numbers, dot product and cross product
2. Define operators for the String class: + (add string or character), type conversion, input/output, [] (get element)
3. Write an Array class for dynamic arrays with operators: += (add element as concatenation), [] (get element) , output an array